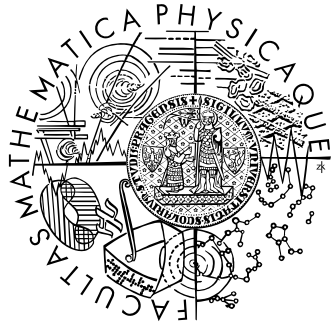
Charles University in Prague
Faculty of Mathematics and Physics

Bachelor's Thesis



Branislav Repček

Distributed regression testing

Department of Software Engineering

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Study Program: Computer Science, Programming

2009

I hereby declare that I wrote this bachelor's thesis on my own and listed all used references. I agree with making the thesis publicly available.

In Prague, December 9th 2009 Branislav Repček

Contents

Abstrakt

Název práce: Distributed regression testing
Autor: Branislav Repček
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Petr Hnětynka, Ph.D.
E-mail vedoucího: petr.hnetynka@mff.cuni.cz
Abstrakt:

Testování je důležitou součástí vývoje softwaru. Cílem automatických testovacích nástrojů je co nejvíce usnadnit testování komplexních aplikací.

Regresní testování je technika testování, při které se testovaná aplikace opakovaně spouští pokaždé, když je aplikace vylepšena. Cílem regresního testování je vyhledávání nechtěných regresí po dobu vývoje aplikace.

V této práci je použit projekt Benchmarking Environment jako základ na vybudování automatického nástroje na regresní testování distribuovaných aplikací. Tato práce je zaměřena na rozšíření projektu o vlastní skriptovací jazyk a nástroje potřebné pro vývoj nových testů. Dále jsou v práci analyzovány výsledky získané pomocí nových testů.

Použitelnost vyvinutého nástroje je demostrována na implementaci regresního testu komponentové aplikace pro SOFA2.

Klíčová slova: *testování software, distribuované testování, regresní testování*

Abstract

Title: Distributed regression testing
Author: Branislav Repček
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail: petr.hnetynka@mff.cuni.cz
Abstract:

Testing is an important part of the software development. The aim of automatic testing tools is to simplify the testing of complex applications as much as possible.

Regression testing is a process in which selected test cases are run every time the tested software receives an update. The aim of regression testing is to find all unwanted regressions during the development of the software.

We use Benchmarking Environment project as a basis for implementation of the automatic tool with support for testing of distributed applications. This work focuses on extending the Benchmarking Environment with custom scripting language and tools needed when writing new test cases and analysing the results.

We have demonstrated the usability of the solution by implementing a regression test case for SOFA2 component application.

Keywords: *software testing, distributed testing, regression testing*

1 Introduction

1.1 Software Testing

Software testing is one of the most important parts of the software development. It is the process of running the software in controlled conditions in order to find errors or evaluate its suitability for a specific task.

Unlike the physical objects, the software does not suffer from wear and tear, but it can fail in most unpredictable ways [18]. Finding the bugs in any non-trivial software is extremely hard simply because of the number of possible states even the simple program can reach. The testing is therefore usually performed against a set of specific cases designed specifically to trigger errors – *test cases*.

As the complexity of the software increases, test case management and execution time of all tests for given software increases dramatically. To help with problem tracking and discovery, several tools have been developed. Among the most know are bug trackers like BugZilla [2] and MantisBT [5] or various testing tools like JUnit [3] and its derivatives for languages other than Java, HP Quality Center [14] and so on.

Testing of distributed applications is particularly demanding. The test cases have to account for possible environment failures outside of the application (e.g. network error) as well as monitor the tested application. With the increasing popularity of the distributed applications and their rapidly growing complexity, the demand for automatic testing tools is increasing.

1.2 Regression Testing

Regression testing is a popular testing strategy as it allows monitoring or the performance of the development teams in the business environment. The main goal of the regression testing is to find regressions in the developed software. Such regressions may be introduced as new features are added to the software, when other bugs are fixed or are simply introduced by the changes in the environment in which the software is executed (e.g. newer version of operating system or a library etc.).

The regression testing requires that the target application is re-examined periodically as it is being developed. As soon as new problem is found a new test should be added to the regression suite to ensure that the problem is not reintroduced to the software at a later stage in the development.

The periodic rerunning of complex applications can require a lot of resources – especially the testers' time which is quite expensive. To lower the costs, the pressure to develop or improve automatic tools which aid in regression testing of the software is increasing.

1.3 The Goal of the Thesis

Because of the complexity of the testing of distributed applications, we believe that the automatic testing tools which assist with the development and deployment of the test cases for distributed applications are needed.

The main goal of this thesis is to design and develop an automated testing tool with support for regression testing of distributed applications. As a basis for our implementation we will use Benchmarking Environment project [15][16] which already contains its own distributed runtime which can run and monitor other applications in heterogeneous environment.

To demonstrate the feasibility of the proposed solution, we will implement a test case for an application running in SOFA2 component system [6][10]. The proof of concept test should support both ad-hoc tests as well as regression tests running automatically on a set schedule.

2 Benchmarking Environment

2.1 Overview

Benchmarking Environment – or BEEN in short – is a tool designed specifically to allow execution of distributed benchmarks in heterogeneous distributed environment [15][16].

To achieve the design goal without sacrificing usability of the environment, the Benchmarking Environment itself is built as a distributed application and is split into two parts with distinct responsibilities – *execution framework* and *benchmarking framework* [16].

2.2 Execution Framework

Execution framework is a generic framework designed to allow execution of tasks in distributed environment. It has been designed to hide differences between various operating systems and to provide unified and easy-to-use interface to the applications which are built on top of it.

Basic unit of execution in BEEN is called *task*. Task can be viewed as a stand-alone application that runs in distributed environment. BEEN differentiates between two different kinds of tasks – *jobs* and *services*.

Services are long running tasks that usually provide some service to other tasks which are part of an application being executed. For example, database or web server may be wrapped in a service. Execution framework provides tools for service management which includes service discovery, status reporting and more.

Jobs are tasks which are meant to execute an action and should end as soon as the action is finished. For example downloading data from source repository (e.g. SVN) is suitable to be a job although it is certainly possible to have "SVN checkout service".

Execution framework consists of 4 main components: *Host Runtime*, *Task Manager*, *Host Manager* and *Software Repository*.

Host Runtime is a component which has to be run on every host participating in the environment. It is responsible for management of tasks running on the host. It also provides logging support with forwarding to a central storage; proxy for communication with other parts of the BEEN and performance monitoring for the host via integrated Load Monitor.

Task Manager is responsible for task scheduling, task dependency management and synchronization. Only one instance of the Task Manager can run in an environment at any given time.

Host Manager is a service which manages database of all hosts in the environment and their hardware and software features. All host utilization data collected by Load Monitors in the environment is stored for each host. Host Manager also provides interface which allows other components or tasks to query the hosts based on their hardware or software specifications. The queries can be specified using a simple query language – Restriction Specification Language (RSL) – which has been developed specifically for this task.

Software Repository is a service which provides storage for all software run by execution framework. The Software Repository uses custom package format – BEEN Package – to store the data together with the metadata describing each package and its contents. Separate query interface with support for RLS queries which allows package lookups is provided.

2.3 Benchmarking Framework

Benchmarking framework is built as an application that runs on top of the BEEN execution environment. The framework supports two different kinds of benchmarks – ad-hoc benchmarks (*comparison benchmarks*) as well as automatically performed regression benchmarks [15][16].

The benchmarking framework consists of two services – *Benchmark Manager* and *Results Repository*.

Benchmark Manager maintains all information required to run a benchmark. It allows scheduling of regression benchmarks and management of past benchmarks. Its architecture is extensible via plug-in based system which allows easy development of benchmarks for new types of applications.

Results Repository is central storage of all results generated in the course of benchmark's execution. Results Repository stores raw data produced by the benchmarks and provides facilities for statistical analysis and visualization of the results. Statistical analysis and evaluation of results is performed using the R language and requires working R installation [19].

2.4 Web User Interface

Important part of BEEN is the Web User Interface. It is a web application which provides user with means of monitoring and controlling activities within the environment. It allows users to quickly review status of the environment without needing any specialised applications except the web browser which can be considered standard on any modern operating system.

2.5 Terminology

The Benchmark Manager uses its own abstraction of the processes in the environment [16]. This abstraction defines a hierarchy with five levels.

- Analysis
  - Experiment
    - Binary
      - Run
        - Tasks

The first four levels of the hierarchy above are referred to as *benchmark entities*. Each entity can store unlimited number of the entities from the level directly below. All entities and tasks have to be registered with Benchmark Manager before the benchmark starts.

The *analysis entity* serves as a container which groups multiple related experiments. Benchmark Manager uses two different types of analyses - *comparison analysis* and *regression analysis*.

- Comparison analysis should contain experiments which are aimed to compare the same software under different configurations or which compare two different products with similar functionality.
- Regression analysis aims to monitor performance regressions of the software caused by its development and is executed automatically according to the given schedule.

The *experiment entity* is a basic unit of the benchmark. It covers all the tasks required to successfully execute the benchmark – for example downloading the benchmarked software, its compilation, deployment and execution.

The *binary* and *run entities* help with avoiding of random effects by compiling and running benchmarked software multiple times. Binaries aim to lower the influence of non-deterministic compilation models (e.g. C++ compilation can result in different binaries with the same code) while multiple runs of the same compiled binary ensures that the random fluctuations in the performance of the environment (e.g. network performance changes, random utilization spikes etc.) do not skew the results.

Each host in the experiment has to belong to a *role*. Role categorizes hosts based on the task they perform. For example, if the experiments needs 3 client servers (physical machines), the plug-in can register a role which requires 3 machines and can specify the hardware and software configuration requirement all hosts in the role have to comply with. When new experiment is created, the roles and pre-selected hosts for each role are displayed as part of experiment's configuration.

3 SOFA2 Overview

3.1 Introduction

As the complexity of the software being actively developed grows, the component-based development has been increasing in popularity as a way of building software systems [21].

There are several definitions of what a component is. Most of them agree that component is a black-box entity with predefined interface and behaviour which can be deployed independently on other components in various contexts [21][24].

The life-cycle of a component, its composition and other rules and features are referred to as a *component model*. The implementation of the component model on a specific environment is *component system* [10].

The idea of component-based development gave rise to number of different component systems in industrial as well as academic environments. The industrial components are aimed at providing a stable environment and usually do not support advanced newly-developed features. The most common industry component systems include Microsoft's DCOM [17] or Sun's EJB [20]**Error! Reference source not found.**. The academic systems usually attempt to provide richer feature set with more powerful designs supporting for example hierarchical component models, behaviour specifications, etc. The systems developed in academy include for example Fractal [7], SOFA2 [6] and others.

SOFA2 is a component system employing hierarchically composed components [6]. Its features include distributed runtime environment, dynamic reconfiguration, multiple communication styles via connectors, behaviour specification via behaviour protocols and more [6][10]. SOFA2 supports all stages of the component application development from component modelling and development to application deployment and execution.

3.2 SOFA2 Runtime Architecture

The SOFA2 itself is a distributed application. The basic abstraction unit is *SOFAnode* [6]. SOFAnode is a distributed runtime environment which may run on several computers and provides basic infrastructure services to its components. Each SOFAnode requires one instance of *repository*, one instance of *dock registry*, one instance of *connector manager* and at least one instance of the *deployment dock*.

Repository

The repository stores both component meta-data as well as component implementations. Only one instance can be running in SOFAnode at any time. It is accessed remotely by other components and used through the whole application life-cycle as the central source of components' descriptions and code base [10].

Deployment Dock Registry

As its name suggests, deployment dock registry is a central registry of all deployment docks in the SOFAnode. Dock registry is used every time one dock needs to communicate with other docks – the registry provides the required look-up services in this case. Only one instance can be running in given SOFAnode.

Global Connector Manager

Global connector manager is responsible for connecting units of connectors together. Only one instance can be running in SOFAnode.

Deployment Docks

Deployment dock can be imagined as a container in which all components are launched. It provides necessary API for starting, stopping and updating of components. Multiple docks can be running in the SOFAnode. Docks are identified by their name.

During application deployment, the docks are selected based on the information stored in application's *deployment plan*. Deployment plan contains all information required to assign specific application component(s) to specific dock(s).

3.3 Running SOFA2 Application

After the SOFA2 application has been developed, it can be executed in the SOFAnode. Following steps have to be taken to run the application if the SOFAnode is not yet running:

1. Start SOFAnode
    1.1. Start repository
    1.2. Start dock registry
    1.3. Start global connector manager
    1.4. Start all docks required by the application's deployment plan
2. Start the application (for example via command-line utilities)

4 Goals Revisited

In previous chapters we have provided the overview of the Benchmarking Environment tool which will be used as a basis for the implementation of the testing tool and we have provided overview of the SOFA2 distributed runtime we will use in a proof of concept implementation of the testing tool.

We have reviewed the architecture of the Benchmarking Environment and reformulated main goals of the thesis according to our findings:

- Extend Benchmarking Environment with support for distributed regression testing. The resulting tool should be flexible enough to allow for different testing strategies without requiring additional modifications to the environment.
- Develop custom language which will allow users to define the flow the experiment. The language should be easy to understand to everyone with at least basic programming background and should allow users to use all the features of the Execution Environment in BEEN.
- Implement a proof of concept regression test for SOFA2 and evaluate usability of the developed solution by reviewing the collected test results.

5 Designing the Solution

5.1 Extending Benchmarking Environment

There are several possibilities of extending the BEEN project with support for the distributed testing. Following approaches have been considered in the design phase of the final solution:

- Develop a new BEEN service similar to the Benchmark Manager. The new *Test Manager* would mirror the architecture of the Benchmark Manager and would allow test authors to use custom language when writing plug-ins.
- Develop a plug-in for the Benchmark Manager. The plug-in would allow test authors to write the test code in custom language and would serve as a bridge between BEEN environment and the TDL code.

Each approach has its own advantages and disadvantages. Both of them share the need for the custom language development, although the resulting language in both cases would be different as it would have to follow slightly different specifications and requirements.

Developing Test Manager Service

The obvious advantage of this approach is that it would allow support for advanced techniques especially suited for software testing with relative ease as an extensive API can be provided by the service.

The resulting design would allow for clean separation of benchmarking and testing parts of BEEN project. This would help new users when learning the tool as it would not mix terminologies and would require less knowledge about the overall design of the whole tool.

However, this design also suffers from some serious drawbacks. Most notably, the increased complexity of the whole solution and the need for relatively substantial changes in some of the services in BEEN (for example Results Repository can only support one data source – the Benchmark Manager).

The code complexity increase would mainly come from the fact that the new Test Manager would need to replicate the plug-in based architecture and scheduler similar to what is currently implemented in Benchmark Manager. Even though both parts can be abstracted and reused, the changes to whole environment are still quite significant.

Developing Benchmark Manager Plug-in

The main advantage of this approach is that its complexity is much lower than the code complexity of the full service. Benchmark Manager has a well-defined interface the plug-ins use and minor interface changes can be done to allow for more functionality without substantially affecting remaining BEEN components.

The obvious disadvantage is the mixing of “two worlds” – benchmarking and testing which would mean that the resulting solution would not be as clean as the solution designed according to the first approach.

Selecting the Approach

After weighing the requirements and benefits of both approaches outlined above, we have decided to implement the second solution. Its development should be faster due to its lower complexity and the resulting tool should still be generic enough to support practically any kind of distributed application. Since the plug-in is using only a specific interface to communicate with the rest of the environment, it is possible to rewrite it to a fully featured Test Manager if required in the future.

5.2 Benchmark Manager Plug-in Design

Each Benchmark Manager plug-in consists of several Java classes, plug-in configuration and metadata files and optionally any additional files or libraries which may be required for the correct functioning of the plug-in.

Each plug-in has three main responsibilities. Each one is handled by a different component of the plug-in:

- Experiment configuration – allows user to configure the experiment according to the requirements of the specific benchmark or test. The configuration is handled by the *configurator* component. Configurator presents user with several screens displayed by the User Interface and processes input entered by the user. Configurator can store configuration data as a set of *experiment properties* in experiment's metadata. Experiment's metadata are then available to other plug-in components and are the recommended way of passing the data between the plug-in components.
- Task scheduling – all tasks required by the experiment as configured by the user are created and submitted to the Task Manager. Task scheduling is implemented in *task generator* component. Task generator runs without any intervention from the user as soon as the experiment is configured.
- Experiment versioning – for each regression benchmark or test it is important to generate new experiments at appropriate time. This is done by *version provider* component which runs periodically based on the schedule specified by user and creates new experiments if necessary (e.g. when new version of the tested software is found). The version provider creates experiments based on a model experiment created by the user during configuration phase. The version provider is not run for comparison analyses since they are aimed for one-off immediate execution without scheduler.

The analysis workflow is different for each type of analyses. For comparison analyses the workflow is relatively simple:

1. Configure the experiment.
   1.1. Few common basic screens are presented to the user. These screens allow user configure the common properties of the experiment – experiment name, required plug-in etc.
   1.2. Configurator from the selected plug-in is started and guides user through appropriate screens for given experiment type. The content of these screens depends entirely on the plug-in.
   1.3. Additional common screens are presented to the user (e.g. assignment of specific hosts to each role).
   1.4. Configurator creates experiment's metadata based on the information entered by the user.
2. Task generator is started and schedules all tasks required by the experiment. Generator has access to the experiment's metadata created by configurator.
3. Benchmark Manager registers all the tasks and manages all further steps of the experiment.

For regression analysis the process is more complex since the analysis typically runs during long time periods and creates several experiments:

1. Configure the model experiment.
   1.1. Common basic screens are presented to the user. Experiment name, plug-in and similar basic properties can be configured by the user.
   1.2. Configurator from the selected plug-in is started and presents user with the screens required to configure the model experiment.
   1.3. Further common screens are presented to the user. In addition to the screens required for the comparison benchmarks, an additional experiment scheduling screen is shown. This screen allows user to configure when new experiments are created and how long the analysis should run.
   1.4. Configurator then creates experiment's metadata based on the information entered by the user.
2. Version provider receives the model experiment and creates as many experiments as required.
3. Task generator is started for each experiment created by the version provider and schedules all tasks required by the experiment. It only receives fully configured experiment from the version provider and does not have access to the model experiment.
4. Benchmark Manager registers all the tasks of all experiments created by the version provider and manages all further steps of the experiment. Experiments are started in batches and processed as fast as possible.
5. Benchmark Manager waits for next scheduled execution of the experiment. When the execution time is reached, the Benchmark Manager will continue with the step 2 – the version provider. Experiments are created in this way as long as has been specified by the user when the analysis has been created.

As can be seen from the above, the main difference between regression and comparison analyses lies in the usage of the version provider which may create several experiments every time it is started (but may also create none).

5.3 Distributed Testing Plug-in Design Overview

The distributed testing plug-in consists of two main parts – plug-in interface required by the Benchmark Manager and implementation of custom scripting language.

The plug-in interface is relatively simple – it implements methods required by Benchmark Manager and only adds few new methods which are used for communication between script interpreter and plug-in components. Most of the work is done in the TDL interpreter as main methods of each plug-in component simply delegate their responsibilities to the interpreter.

Distributed Testing Configurator

The configurator in distributed testing plug-in only presents two custom screens to the user. On the first screen, number of binaries and runs has to be specified, the second screen is used to enter the TDL code of the test case.

Configurator part of the TDL code is interpreted immediately after the code is submitted by Web Interface to the Benchmark Manager.

Distributed Testing Task Generator

The generator is the most complex component of the plug-in. The actual submitting of the experiment's tasks is handled by the TDL interpreter; however, task generator has to automatically generate result collection tasks and tasks which upload results to the Results Repository.

Generator's TDL code is executed after user confirms experiment's details in the Web Interface. After generator is finished, complete list of all tasks executed within the experiment is presented to the user.

Distributed Testing Version Provider

The version provider is relatively simple. All the modifications of the model experiment are performed by TDL code. Currently the version provider does not implement any advanced algorithm for version tracking past the default algorithm provided by the Benchmark Manager.

Version provider's TDL code is executed every time the Benchmark Manager's scheduler starts regression experiment (it is not executed for comparison experiments).

The general plug-in design and experiment workflow outlined above imposes one severe restriction on the way the tester plug-in can interact with the environment – all the code from plug-in is executed before any of the tasks in the experiment are actually started.

This means that the plug-in will not be able to react based on the immediate state of the execution environment during experiment's execution. However, this restriction does not limit the overall functionality of the plug-in since BEEN provides other facilities for task synchronisation in form of checkpoints and task dependencies which

are maintained by Task Manager and Host Runtime. In cases where the built-in functionality does not suffice, it is possible to develop custom tasks which have access to whole BEEN environment.

5.4 Test Results Processing

The main difference between tests and benchmarks lies in the way their results are interpreted. Benchmarks typically produce a set of numbers which assess the relative performance of the application in a specific way (for example table with "ping" times to specific destinations, memory usage and so on). On the other hand, tests usually produce simple boolean results – test passed or test failed.

For benchmarks, BEEN is using relatively simple algorithm when processing the results:

1. The results are created by one task in benchmark – *the benchmarking task* – in whatever format is suitable for given benchmark. This task has to be written specifically for each new benchmark.
2. During experiment creation, task generator creates special *check and convert task* which reads the data generated by task from the first step. The data is validated and converted to NetCDF [23] format which is used internally by Results Repository as a data container. The validation and conversion task has to be implemented according to a specific template which makes the development of such task much easier (more precisely, this task is simply derived from an abstract class and only two methods need to be implemented).
3. The results are uploaded to the Results Repository by a *log collection task*. Log collection task is common for all benchmarks. This task is part of the default installation of the BEEN environment.
4. After the results are collected, the Results Repository can run a set of specialized scripts written for R [19]. These scripts can calculate custom statistics based on the benchmark results. The scripts are initially set in configurator and can be later configured by user. Since the scripts require knowledge of the benchmark's data, they have to be tailored to specific benchmark.

The above algorithm requires that two tasks are written for each new benchmark. One new task is the *benchmarking task* which runs the benchmark and the other one is the *data validation and conversion task* which reads the benchmark's data.

Since the test results are simpler in structure, we have developed a common data validation and conversion task which has to be used by all tests. This task expects test results in simple text file called `results.txt` located in the working directory of the testing task. On each line of the `results.txt` file, two values separated by colon are required – name of the test and its result (0 for failed, 1 for successful test).

For example, the valid results file may look like this:

```
start_up:1
output_correct:0
no_exceptions:1
shut_down:1
```

Test names (the first field on each line) are translated to column names in Results Repository when results are collected at the end of the experiment.

All the result collection tasks and their dependencies are automatically generated by the task generator after author of the test marks the tasks which collect results via `register_result_task` function in TDL code.

5.5 Additional BEEN Improvements

Apart from writing custom Benchmark Manager plug-in to support the testing, several new features have been added to the BEEN to ease the plug-in development and improve existing functionality.

Passing Checkpoint Values as Property Values

It is now possible to use references to checkpoint values from other tasks when defining task's properties or class path. The references use format similar to variable references common in shell scripting languages `${task_id:value}` where *task_id* is TID of the task from current context and value can be one of *taskDirectory*, *workingDirectory*, *temporaryDirectory*, *hostName* or name of the checkpoint. Checkpoint name can be any string; however, it cannot contain right curly brace or colon characters.

The reference is resolved in the moment task is started and its occurrence is replaced by the actual value. It is only possible to refer to the scheduled tasks in the same context. The reference correctness is only checked when the task is started.

In the proof of concept test we use this feature for example to let the Ant build tasks know where the SOFA2 source code is located since it is stored in the working directory of task which downloads it from SVN repository.

Custom Class Path for Tasks

Ability to add class path entries to scheduled tasks has been added to Task Manager and Host Runtime. It is now therefore possible to have one task download specific library and use it in other task without having to build new task package.

Class path can only be defined when task is being scheduled. When the task is started, the new class path entries are appended to existing class path defined in the tasks package.

Simple Resource Allocation Scheme for Benchmark Manager

To prevent multiple experiments colliding with each other a very basic resource allocation scheme has been added to the Benchmark Manager.

When an experiment is scheduled a set of resources required by the experiment can be specified in experiment's metadata. Resources are identified by their name (a simple string). When the experiment is started, the Benchmark Manager looks into the list of currently used resources and if all the resources required by the experiment are free, the experiment is allowed to continue. If some of the resources are already, the experiment is put into a queue and will be started after all resources are available. After the experiment is started, all its resources are marked as used until the experiment ends.

The resources are shared between all analyses running in the BEEN environment. It is therefore important to not use too generic resource names which may collide with unrelated experiments.

Even though the algorithm is very simple, it can be used in regression experiments to make sure that multiple experiments requiring the same resource (e.g. host:port pair, external device) are not started at the same time. This feature is used in the proof of concept test to make sure only one SOFA runtime is running at a time.

Results Repository Improvements

Support for more complex statistics display for all entities of the benchmark/test has been added to the Results Repository – it is now possible to calculate, store and display multiple tables with different dimensions for each entity while in the original version this feature was only available for runs.

6 Test Definition Language

During analysis of the possible experiment workflow, we have come to the conclusion that to improve the usability of the BEEN and make the test development easier we have to design and implement custom language in which the test cases would be designed. We dubbed this language *Test Definition Language*.

6.1 Basic Design Principles

Several key properties of the Test Definition Language (TDL) have been identified when designing and analyzing its interactions with the rest of the environment:

- The language has to be generic enough to allow for writing of complex tests for applications with large number of components. Therefore the language should support usual programming constructs like variables, math expressions, conditionals and so on. Support for "advanced" features like exceptions, custom subroutines or threading is not required.
- The language has to be easy to learn for anyone who has at least some programming background. This would mean that the syntax of the language should resemble the syntax of commonly used programming languages. In our case we used ideas from C++ [22], Java [13] and ECMAScript [11] when designing the grammar.
- The implementation should provide simple way of adding more functionality should the built-in functionality be insufficient for a specific case. Again, the aim is simplicity of the solution, so there is no need for implementing extra plug-in system just for TDL.
- Language should only allow safe operations in each phase of the experiment's workflow. That is, it should not allow user to run tasks when the configurator is running and it should never allow the TDL code to crash Benchmark Manager since it runs within its process.
- The language will be interpreted as the interpreter is relatively easy to design and there is no need for high performance in this case. The execution time of the TDL script is insignificant for any practical test compared to the time and resources required when running the distributed application with its own runtime.

6.2 TDL Syntax Overview

As mentioned in previous section, we have decided to model syntax of the TDL according to the syntax of popular languages like Java or C++.

TDL is an imperative language with source file structure similar to the files in C++ or Java. Identifiers in TDL are case-sensitive and can contain letters, numbers or underscore characters. Identifiers cannot start with a number. Leading underscores are allowed, but discouraged due to their lower readability.

Code Blocks, Stages and Entities

The structure of the TDL source file closely mirrors the structure and phases of the experiment in BEEN. The source file is organized into a tree-like structure which consists of *stage* and *entity code blocks*.

Code block is a block of code enclosed between matching curly braces ("{" and "}"). Each code block defines a new scope for variables and constants.

Stage is a named code block which corresponds to specific component of the Benchmark Manager plug-in. Stages can only appear at the level 1 of the AST (level 0 being the whole source file).

It is not possible to define custom stages – only three stages are allowed: *configurator stage*, *generator stage* and *version provider* stage.

Stages are further refined by *entity code blocks*. Entity code block is a named code block which can only appear as a direct child node of a stage node in AST of the TDL source file. Each stage has a different set of entities with different semantics.

Configurator stage

Configurator stage is required in each source file. The configurator stage runs after the user has entered the TDL code in the Web Interface. It allows TDL script to define new roles required by the experiment. Configurator stage only has one mandatory entity – `main`. No optional entities are supported in configurator stage.

```
#config {

    main {
        // The configurator code goes here
    }
}
```

Generator stage

Generator stage is required in each TDL source file. It is run every time the generator runs for new experiment. It has to contain three mandatory entity code blocks – *experiment entity* block, *binary entity* block and *run entity* block. Each entity block corresponds to Benchmark Manager entity with the same name. Analysis entity does not have associated entity code block since it is not possible to run any tasks at analysis level.

```
#generator {

    experiment {
        // Code executed for experiment entity
    }

    binary {
        // Code executed for each binary entity
    }

    run {
        // Code executed for each run entity
    }
}
```

Version provider stage

Version provider stage is only required in regression tests. For comparison tests it can be omitted as the Version Provider is not used.

It contains only one entity which is executed every time the version provider runs.

```
#version {

    main {
        // Main body of the version provider code
    }
}
```

Variables and Constants

Variables and constants in TDL are not typed and are declared with keywords `var` or `const`. It is possible to combine declaration and definition of a variable into a single statement. For constants the definition is mandatory.

```
var name;
var name2 = "value";
const CONSTANT = 12345.6;
```

TDL variables and constants can store values of following data types:

- `Boolean` – boolean value which can be either `true` or `false`.

  ```
  var x = true;
  ```

- `Number` – floating-point or integer number. Integers are stored as 64-bit `Long` numbers. Floating-point numbers are stored as 64-bit IEEE 754 values (`Double` in Java). The storage format is determined based on the number being stored – if it is an integer, it will be stored as an integer. If the number cannot fit into an integer, if it contains fractional part, or is written using scientific notation, it will be stored as a float.

  ```
  var integer = -12345;
  var float = 4.28e-3;
  ```

- `String` – Unicode strings of any length. Strings have to be quoted with double quotes and the same escape sequences as in C++ are supported.

  ```
  var x = "this is a string";
  var y = "this is a \"string\" with\tescape sequences";
  ```

- `List` – generic list which can contain elements of multiple types. The elements of the list can be of any type (even lists) and different types can be mixed in one list. List indexing starts at zero and the last element has index `length - 1`.

  ```
  var simple_list = [1, 2, 3];
  var empty_list = [];
  var mixed_list = [1, -2.5, "abcd", [1, 2], false];
  ```

By default TDL variables are strings unless value of different type is assigned to them. The default value for variables without explicit initialization is an empty string.

Implicit conversions between types are never done except for the operands of string concatenation operator – if one of the operands is not a string, it will be converted to string using the default `toString` method of given data type..

Variables and constants in TDL are only accessible in the scope in which they are defined or in child scopes of their defining scope. Top level scope is called the global scope and begins before the first character of the source file and extends just past the last character of the source file. Stages, entities and code blocks (including loop bodies and if-then-else branches) all define new scopes.

TDL allows variable name shadowing – variable or constant from given scope can shadow the variable or constant of the same name from its parent scope. Built-in variables or constants (e.g. `CURRENT_RUN` or `BINARY_COUNT`) cannot be shadowed.

Conditional statements

TDL supports if-then-else statements with similar structure as in Java or C++. The basic syntax is following:

```
if (condition) {
      // Then branch code
}

if (condition) {
    // Then branch code
} else {
    // Else branch code
}
```

Then and else branch statements have to be written as code blocks except when the branch is another if-then-else statement which can be used directly.

```
// Valid simple if-then
if (condition) { code; }

// Error – missing code block
if (condition) statement;

// Valid statement
if (condition1)
    if (condition2) { code; }

// Also valid statement
if (condition1) {
    code;
} else if (condition2) {
    code;
}
```

Loops

TDL supports all usual loop constructs with following syntax:

- C++-style for loop

  ```
  for ([initializer]; [condition]; [update]) {
      // Loop body
  }
  ```

It is possible to leave out initializer, loop condition or loop update statements. Definition of new variable is also allowed in initializer statement. Variable will be visible only in the scope of the loop body and in the condition and update statements:

```
for (var name [= value]; [condition]; [update]) {
    // Loop body
}
```

- C++-style while loop

```
while (condition) {
    // Loop body
}
```

- C++-style do-while loop

```
do {
    // Loop body
} while (condition);
```

The loop body has to be always written as a code block – simple statements are not allowed.

Function Calls

It is possible to call various built-in functions from the TDL code. TDL does not support definition of new custom functions in TDL code.

The functions in TDL belong to two different classes – *regular* and *modifier functions*. Regular functions behave just like functions in Java or C++. Modifier functions are specialized functions which can only be called when they are being passed as an argument to a regular function. As their name suggests, modifier functions modify how their parent function behaves. Each modifier function is tightly bound with its parent function – in fact, trying to call modifier function in wrong parent context will result in a parser error.

Modifier function support was introduced into TDL to support passing of complex data structures to TDL functions without requiring TDL itself to support such structures (either as simple C-like `struct` or more complex objects).

Function call in TDL consists of a function name followed by arguments enclosed in round parentheses:

```
function_name(argument1, argument2)
```

Arguments are separated by commas. Functions which do not take any arguments have empty argument lists. Call of a modifier function has to be preceded by "@" character:

```
@modifier_function(argument1, argument2)
```

It is not possible to call modifier function on its own and use its return value in an expression (it cannot be assigned or copied). Modifier function always needs to be called in the context of its parent function directly as one of its arguments.

Internal full name of the modifier function consists of name of the parent function followed by "@" and the name of the modifier function. This is very similar to name mangling performed by C++ compilers to differentiate member functions which belong

to different objects. For example, the `run_task` function allows `task_wait` modifier as one of its parameters. Full name of the modifier function is then `run_task@task_wait`.

Regular functions in TDL always return a value – there is no equivalent of `void` functions from Java.

TDL does not support definition of custom functions from within TDL scripts. If new function is required, it has to be written in Java and properly registered with TDL parser. To this end several utility classes which aid in development of new functions have been developed. All new functions have to be implemented as derived classes of `AbstractFunction` class. To register the function with the parser it can simply be annotated with `TDLFunction` annotation.

Notation

In the rest of the text we will use following notation when introducing new functions available from TDL code:

```
return_value_type function_name(type1 arg1, [type2 arg2 = default_value])

@modifier_function_name(type1 arg1, [type2 arg2])
```

Where *function_name* and *modifier_function_name* are the function names; *return_value_type* and *typei* are types mentioned in the *Variables and Constants* section above. Additionally, we will use *Any* to denote that given argument can be of any type; *Integer* if integer number is required (e.g. as an index in a list) and `List<`*elem_type*`>` to denote that argument has to be a list with all elements of type *elem_type*. Note that these types are only used in the text of the thesis and cannot be used directly in TDL code.

Optional arguments will be enclosed in square brackets. For each optional argument the *default_value* is the value the argument will have if it is not specified in the argument list. If any number of arguments can be used we will use ellipsis ("...") to signify that all following arguments can have the same type as the last argument.

Following function signatures demonstrate the notation:

```
Number int(Number x); // Return integer part of given number.
String to_string(Any x); // Convert value to string.
Integer list_length(List list); // Return length of the list.

// Select role (and host) in which task will be executed.
run_task@role(String role_name, [Integer host_index]);

Number min(Number... x); // Find minimum from given numbers
Number min(List<Number>... x); // Find minimum from all the lists
```

6.3 TDL Parser

The parser for TDL language has been developed using JavaCC parser generator library [3]. JavaCC has been selected because it is relatively easy to integrate resulting parser into any Java code and the library is already used in BEEN to generate parser for *Restriction Specification Language* (RSL) which is used in Software Repository and Host Manager.

TDL source code is interpreted as a stream of tokens with tokens separated by white space characters (space, tab character, new line). During parsing, the TDL parser builds abstract syntax tree (AST) of the source code. The parsed AST can be passed directly to the interpreter in later stages of the experiment processing.

TDL parser does not perform any recovery when error in the parsed source is encountered. The parsing therefore stops as soon as the first error is found. We opted for this solution since error recovery is complex feature which would increase development and testing time.

One of the design goals of the parser was to provide the error messages which contain enough information for the test author to effectively find and fix the problems which may occur during the test case code. This greatly improves the usability of the language since it is not possible to directly debug the code from Web Interface.

6.4 Interpreting TDL Code

TDL interpreter is designed as a simple visitor which walks the nodes of the AST generated by the parser. The state of the interpreter is determined by the nodes it has processed and its *interpreter context*.

Interpreter context stores values of all variables and constants, current stack and provides access to the rest of the BEEN environment via adapter class. This design allows development of custom replacements of various BEEN classes (mainly benchmarking plug-in parts) for testing purposes (e.g. `DummyGenerator` class used in JUnit tests for the parser and interpreter).

The basic execution unit of the TDL code is the stage. Each stage has its own semantics and contains different entities and therefore it is interpreted in a different way. However, the general algorithm used when interpreting any stage is quite simple:

1. Define all built-in global constants.
2. Interpret all constants defined in global scope of the source file. Constants are defined in the same order in which they are specified in the source file. Variables are not allowed in global scope (this is enforced by the parser).
3. Start interpreting the stage.
4. Define all variables or constants defined in the stage's scope. All variables and constants are processed in the same order in which they appear in the source file.
5. Interpret entities in the stage according to their semantics. Entities are simple named blocks and are interpreted as regular blocks of code.

Interpreting Configurator Stage

The configurator stage can only contain one entity – `main` entity. This entity is interpreted only once during the configuration of the experiment after user has clicked through all the screens in the configurator. Function in this entity have access to instance of `DTConfigurator` class which allows TDL script to create new roles and assign hosts to them.

Interpreting Generator Stage

Generator stage is the most complex stage since it contains three mandatory entities – `experiment`, `binary` and `run`. Each entity of the stage represents one entity from the Benchmark Manager. Analysis entity is not included since it is not possible to run any tasks at analysis level (analysis is just a container which groups similar experiments).

Entities in the generator stage are the only entities in which it is allowed to schedule new tasks via `run_task` TDL function. Tasks scheduled from given entity block are assigned to appropriate entity in the experiment.
The entities are interpreted as follows:

1. Interpret `experiment` entity. This entity is interpreted only once and tasks created in this entity are executed at the beginning of the experiment.
2. Interpret `binary` entity as many times as there are binaries in the experiment.
3. For each `binary` entity interpret `run` entity as many times as there are runs in the experiment.

The algorithm above will result in (binary count)*(run count) runs in the experiment.

The generator stage is interpreted every time the generator runs. For comparison analysis it is therefore interpreted only once. For regression analyses it is interpreted once for each new experiment created by the version provider.

Functions in generator stage have access to an instance of `DTTaskGenerator` class which provides methods to add new tasks and assign them to specific experiment entities.

Interpreting Version Provider Stage

Version provider stage can contain only one `main` entity. This entity is interpreted once every time the version provider runs. The main entity is the only entity which is allowed to create new experiments and access the model experiment.

Since the version provider is not executed for comparison analyses, this entity is only interpreted for regression analyses.

Functions from version provider stage have access to an instance of `DTVersionProvider` class. This allows TDL scripts to read and modify metadata of new experiments and submit new experiments to the Benchmark Manager.

7 Proof of Concept Test for SOFA2

7.1 Overview

To prove that the solution we developed is actually usable in real-life scenario, we have decided to implement a simple test of a component-based application running in SOFA2 component system.

In this chapter we will demonstrate the techniques required to build working test suite using TDL. At first, we will present very simple synthetic test case which will then be followed by more complex implementation of SOFA2 test suite.

7.2 General Test Requirements and Guidelines

- Configurator and generator stages are mandatory. Version provider stage is required for regression analyses and optional for experiments in comparison analyses.
- Each test has to have at least one role registered in its configurator stage. The roles can be registered with `add_role` function:

```
Boolean add_role(String role_name,
                 String role_rsl,
                 [Number host_count = 1])
```

- Each test has to have at least one role designated as a role in which result generating tasks are run. The role set-up is done in configurator stage with `add_results_role` function:

  The role being registered as result role has to exists (it has to be defined via `add_role`).

```
Boolean add_results_role(String role_name)
```

- Experiments can request specific resources for their exclusive use during the configurator stage. Each resource is identified by its name and resource usage is tracked by Benchmark Manager. The resources can be added with following function:

```
Boolean add_experiment_resource(String resource_name)
```

- To create new experiment in version provider stage user has to clone experiment's model. The cloned experiment can then be modified and submitted to the benchmark manager. Only one copy of experiment's model clone is kept active at a time.

  The general algorithm for experiment creation is then following:

```
Boolean model_clone(); // Create new clone of experiment's model

// Modify current clone with one of the functions below

Boolean submit_experiment(String version); // Submit new experiment
```

Several functions are available when working with the cloned metadata:

```
Boolean has_property(String property_name)
String get_property(String property_name)
String set_property(String property_name, Any new_value)
String set_experiment_name(String new_name)
String get_experiment_name()
```

It is also possible to access list of version which have been submitted to the Benchmark Manager in previous runs as well as list of version already processed in previous experiments:

```
List< String > get_submitted_versions()
List< String > get_processed_versions()
```

- Experiments generated in version provider stage should be ordered according to their version from the oldest to the newest one. It is allowed to submit more than one experiment at a time.

7.3 Working with Tasks

TDL provides extensive interface for submitting new tasks which allows controlling almost all aspects of the tasks' life-cycle.

The only way of submitting a task is via `run_task` function. This function can only be called from `experiment`, `binary` and `run` entities of the `generator` stage. It is not allowed to submit tasks from any other part of an experiment.

Tasks submitted via the `run_task` call will be assigned to the entity from which the call has been made (for example if function is called from `binary`, the task will belong to currently processed binary entity in the experiment).

`run_task` has quite complex function signature and accepts several modifier functions which refine the behaviour of the task:

```
String run_task(String task_name, String package_name,
  @role, [modifiers...])

String run_task(String task_name, String package_name,
  @as_task, [modifiers...])

String run_task(String task_name, String package_name,
  String host_name, [modifiers...])

String run_task(String task_name, String package_name,
  List< String > host_list, [modifiers...])
```

All the signatures above can be used, however it is recommended to only use the first two as they provide the best integration with the Benchmark Manager.

The first signature schedules the task on one of the hosts which have been designated for specific role by the user when configuring the experiment in the Web Interface.

The second one allows submitting the tasks on the same host as the previously scheduled task. This allows creating the task chains which use the same data without requiring tasks to upload/download the data to different host.

The third and fourth signatures submit tasks which will be executed on specified host or on one of the hosts from given list of hosts.

Arguments

`task_name` Name of the task which will identify the task within the environment. Task name can be string of any length and can contain any characters although it is recommended to only user letters, numbers, underscore and dash to make sure tasks are easily recognizable in User Interface. Task name has to be unique within the experiment's context.
To make sure the name is unique it is recommended to use the following built-in constants in combination with the custom task name prefix:

```
//ID of the current entity
String CURRENT_ENTITY_ID;
// Total number of runs
Integer RUN_COUNT;
// Index of current run, 0..RUN_COUNT-1
Integer CURRENT_RUN;
// Total number of binaries
Integer BINARY_COUNT;
// Index of current binary, 0..BINARY_COUNT-1
Integer CURRENT_BINARY;
```

`package_name` Name of the Software Repository package which contains the task's data.

`host_name` Name of the host on which the task should be started. Only the hosts registered in the environment can be used.

`host_list` List of host names on which the task can start. Task Manager will pick one of the hosts when the task is started. The exact algorithm for host selection is unspecified and no assumptions should be made about the specific host being picked.

`modifiers` Any number of arguments which are all calls to one of the following modifier functions:. Each modifier call has to be in a separate argument.

`@role` Call to the `@role` modifier function which specified the role in which the task will run. This is the recommended way of starting the tasks as it allows user to select the concrete host in the User Interface.

`@as_task` Call to the `@as_task` modifier function which will cause the task to start on the same host as the task submitted previously. This should be always used when given task needs to access the data produced by another task running before it (e.g. compilation task needs to access the source code which may be downloaded by another task).

Return Value

Function always returns the name of the task – the first argument. This is convenient especially when the name of the task is "computed" by concatenating several strings/values together (e.g. task name prefix with entity ID).

Modifier Functions

Several modifier functions can be used to refine the tasks behaviour. All of them except `@role` and `@as_task` can be specified fourth and subsequent arguments of the function call. `@role` and `@as_task` can only appear as the third argument.

Following is just an excerpt of the full specification for all the modifier functions. Full documentation is available within detailed comments for `RunTask` class in `RunTask.java` source file.

```
// Make task run in specified role on given host from the role.
@role(String role_name, [Integer host_index = 0])

// Make the task run on the same host as previously scheduled task.
@as_task(String task_name)

// Assign given value to a task's property.
@property(String property_name, Any value)

// Add dependency on a specific task and its status.
@task_wait(String task_name, [Integer wait_for_event = TASK_SUCCESS])

// Add dependency on specified checkpoint reached by given task.
@checkpoint_wait(String task_name, String checkpoint_name,
    [Any checkpoint_value])

// Append given values to the task's class path.
@add_classpath(List< String > new_classpath_entries)

// Mark task as exclusive, context exclusive or default (not exclusive).
@exclusivity(Integer exclusivity_type)
```

7.4 Trivial Example Experiment

To illustrate the usage of the TDL facilities when writing new test case we have developed very simple example experiment. Following code is a trivial test which runs one task in experiment, then one task for each binary and finally one task for each run. Two roles are registered for this experiment – one role is used also for results collection. We use the dummy testing task to generate random results. The test also supports regression testing – one new experiment is generated every time the Version Provider runs.

```
#version {
    main {
        var new_version = to_string(time());

        /* Clone model experiment and set its name. No other properties
           are necessary, so the experiment is submitted right away. */
        clone_model();
        meta_set_name(meta_get_name() . " " . new_version);
        submit_experiment(new_version);
    }
}
```

```
#config {
    main {
        // Following roles will match all hosts
        add_role("first role", "name =~ /.*/", 1);
        add_role("second role", "name =~ /.*/", 1);
        add_results_role("second role");
    }
}

#generator {
    /* Random seed for the result generating task is updated every time
       the task is submitted to get different data for different runs. */
    var tester_seed = time();

    /* Keep task names so that the sub-entity waits for last task from
       its parent entity. */
    var experiment_task;
    var binary_task;

    // Name of the testing checkpoint for binary task.
    const CHECKPOINT_NAME = "test_cp";

    experiment {
        // This task will wait for 5 seconds and exit.
        experiment_task = run_task(
            "experiment-task-" . CURRENT_ENTITY_ID,
            "testworker",
            @role("first role"),
            @property("do.wait", "true"),
            @property("wait.time", 5)
        );
    }

    binary {
        /* Following task will wait for the task from current experiment.
           It will only set checkpoint "test_cp" to "aaa" and exit. */
        binary_task = run_task(
            "binary-task-" . CURRENT_ENTITY_ID,
            "testworker",
            @role("first role"),
            @property("do.checkpoint.set", "true"),
            @property("checkpoint.name", CHECKPOINT_NAME),
            @property("do.checkpoint.custom.value", "aaa"),
            @task_wait(experiment_task)
        );
    }

    run {
        /* This task will wait for previous binary task to set "test_cp"
           checkpoint. After it starts it will generate results of 4
           simulated tests with random data. */
        const test_task = run_task(
            "test-task-" . CURRENT_ENTITY_ID,
            "dummytest",
            @role("second role"),
            @property("test.count", 4),
            @property("test.name.prefix", "Test #"),
            @property("random.seed", tester_seed += 1000),
            @checkpoint_wait(binary_task, CHECKPOINT_NAME)
```

```
        );

        /* Let Benchmark Manager know which task generates results so
           that result conversion can be scheduled with appropriate task
           dependencies. */
        register_result_task(test_task, "second role");
    }
}
```

As can be seen from the above example, the code is relatively lengthy, but still simpler and easier to read than its Java equivalent would be.

7.5 SOFA2 Requirements

As described in the third chapter, SOFA2 requires several runtime components to be started before any SOFA2 application can be run. Several specialized tasks have been developed to run the SOFA2 runtime components and manipulate SOFA2 environment.

Since the runtime components of the SOFA2 are designed as stand-alone applications the design of the supporting tasks is pretty straightforward and none of them are longer than about 250 lines of Java code including comments. Moreover, the tasks which support SOFA2 runtime can be reused in later experiments and therefore much less code needs to be written for new tests.

The tasks and services which host SOFA2 runtime components have been designed so that they do not carry SOFA2 libraries with them. This allows the SOFA2 environment to be downloaded from SVN and compiled directly from source without having to build new task packages which would run new version of SOFA2.

This design decision has of course one drawback – the class path for almost all SOFA2 tasks has to be set manually to contain the downloaded libraries. Considering the number of libraries required for SOFA2, the class path is quite long, but it can be set easily with help of few constants and built-in list manipulation functions.

Additionally, SOFA2 compilation requires working installation of Apache Ant [8] and Apache Ivy [9] on the host which will serve as a compiler. Due to the way Ant is distributed (simple archive without installation package) it not possible to reliably detect presence of Ant on given host via Host Manager's queries. This slightly inconveniences the user of the test since the compiler host cannot be automatically preselected when the hosts are assigned specific roles during the experiment's configuration.

SOFA2 Repository

The repository is wrapped inside the BEEN service named `sofa-repository`. The service does not expose any functions of the repository remotely – it just runs the repository in the background.

If the repository is started successfully, the task will set checkpoint `repository-started` to value "started".

SOFA2 Dock Registry

Dock registry is implemented as BEEN service named `sofa-registry`. As with the repository service, no methods from the dock registry are exposed via remote interface belonging to the service.

Checkpoint `registry-started` is set to value "started" as soon as the dock registry has been successfully started.

SOFA2 Global Connector Manager

Global connector manager is implemented as BEEN service with name `sofa-conman`. Again, it does not expose connector manager's API, but simply runs it in the background.

To signify the successful start of the connector manager the checkpoint `conman-started` to value "started".

SOFA2 Deployment Dock

Deployment dock is again implemented as a BEEN service. However, since it is possible (and quite common) to run multiple docks in one experiment, the name of the service changes based on the name of the dock. This is required since BEEN does not support multiple instances of one service with the same name. The service registers itself as `sofa-dock-`*`<dock_name>`* where *`<dock_name>`* is name of the dock running in the current instance of the service. The actual dock name is read from a task's property which has to be set when the task is scheduled.

After the dock is successfully started, the service sets checkpoint `dock-started` to "started".

SOFA2 Application Controller

Application controller is regular task which can be used to start or stop SOFA2 applications in current SOFAnode.

When this task is started, it will simply connect to the SOFAnode via its Dock Registry and request application start-up or shut down based on its properties.

When the controller is used to start the application it will signify the successful application start by setting the checkpoint `app-started` to the ID of the newly started application assigned to it by the SOFA2 runtime.

When controller is used to stop the application, it requires an ID of the application to stop. After the application has been stopped, the checkpoint `app-stopped` is set to the ID of the application which has been stopped.

LogDemo Tester

This is the testing task which actually generates the result file as has been described in the section *5.4 Test Results Processing*.

This task is designed to test the LogDemo application which is part of the SOFA2 package. The tests it performs are quite basic; however they should still be sufficient as a proof of concept. Four tests are performed:

1. Successful start of the application. This is verified based on the status of the controller task used to start the application in current SOFAnode.

2. Appearance of the message “Hello world!” in the output of the dock which hosts the application.
3. Exception-free run of the application – this is verified by making sure the word “Exception” does not appear in the output of the dock which hosts the application.
4. Successful shut down of the application. This is verified based on the status of the controller task used to shut down the application.

This task does not require SOFA2 libraries and therefore it is not necessary to customize its class path when it is being scheduled.

7.6 Implementing the SOFA2 Test

As a proof-of-concept we have implemented SOFA2 test case which uses the facilities provided by the Distributed Testing plug-in. We have aimed for a relatively simple case which shows the potential of the environment while still being reasonably complex and functional. The proof-of-concept test case should support both comparison and regression tests to showcase the most important features of the developed framework.

The full TDL source code of the test case is quite lengthy and therefore it is only provided in electronic form on attached DVD.

Following is the high-level overview of the processes which need to be carried out in order to successfully download, compile, run and test SOFA2 demo application using our framework. The processes are split into groups which correspond to experiment entities.

Experiment

1. Download SOFA2 source from its SVN repository.

Binary

2. Compile the source package with Ant. This has to wait for the download task from experiment entity to finish.
3. Upload the binary package to the Software Repository.

Run

4. Wait for the compilation and package upload from binary entity to finish.
5. Download the SOFA2 runtime to host which will run the runtime components.
6. Start SOFAnode with correct settings. All SOFA2 services have to be configured start with correct class path which points to newly compiled version.
7. Download the SOFA2 runtime to host which will execute the SOFA2 application.
8. Start SOFA2 dock for nodeA with correct class path and properties.
9. Run LogDemo application in configured SOFAnode started in previous steps.
10. Wait for the application to start-up and give it some time to write some messages to the log file.
11. Stop the LogDemo application.
12. Collect results of the run with tester task.
13. Upload results to Results Repository.
14. If this is the last run of current binary stop the all the SOFA2 services including the dock.

The outline above results in several tasks with relatively complex dependencies being submitted for each experiment. The tasks and their dependencies are shown on following diagrams.

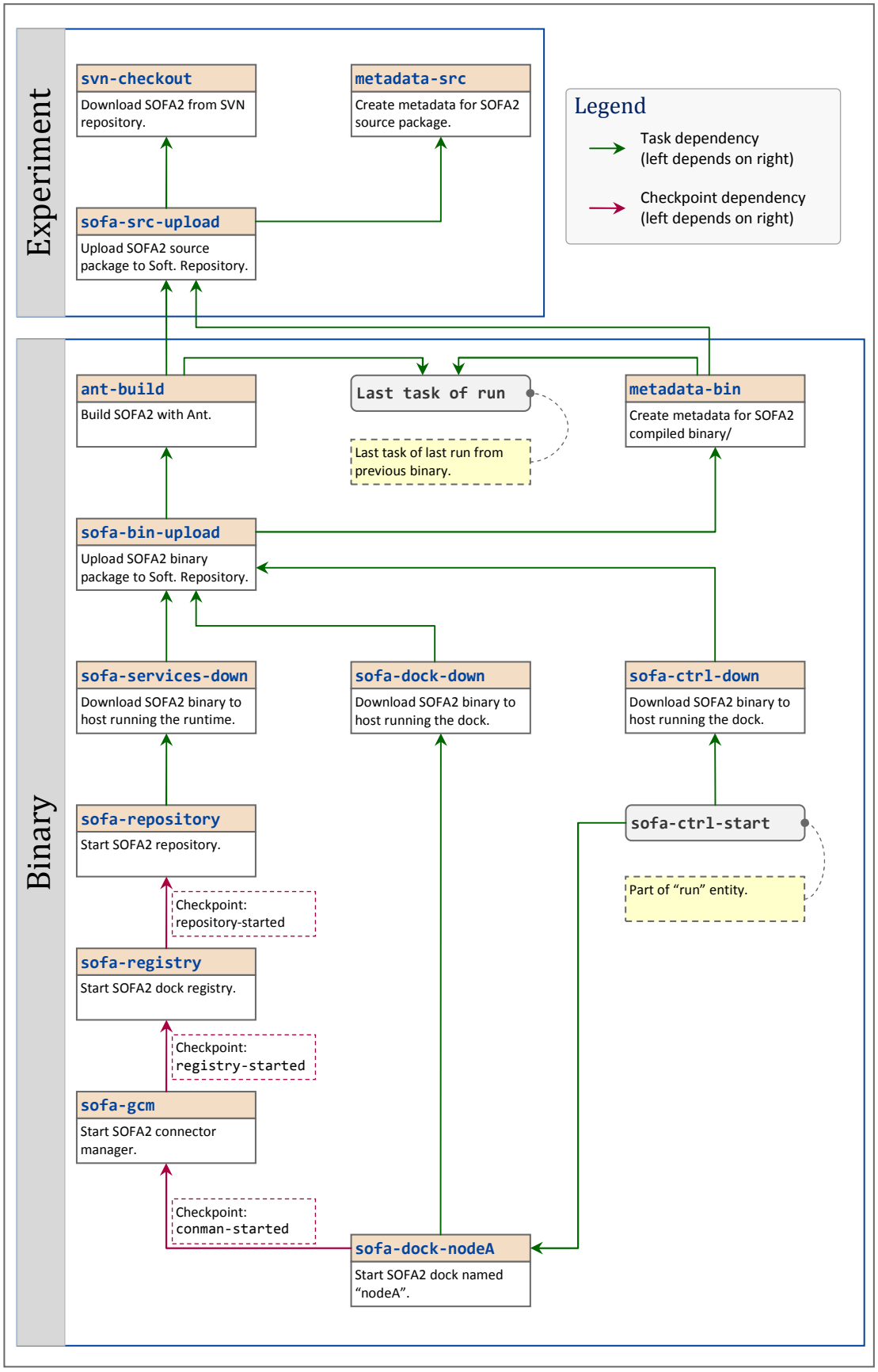


Figure 1: SOFA2 LogDemo test tasks scheme. Following picture only shows tasks which run in experiment and binary entities.

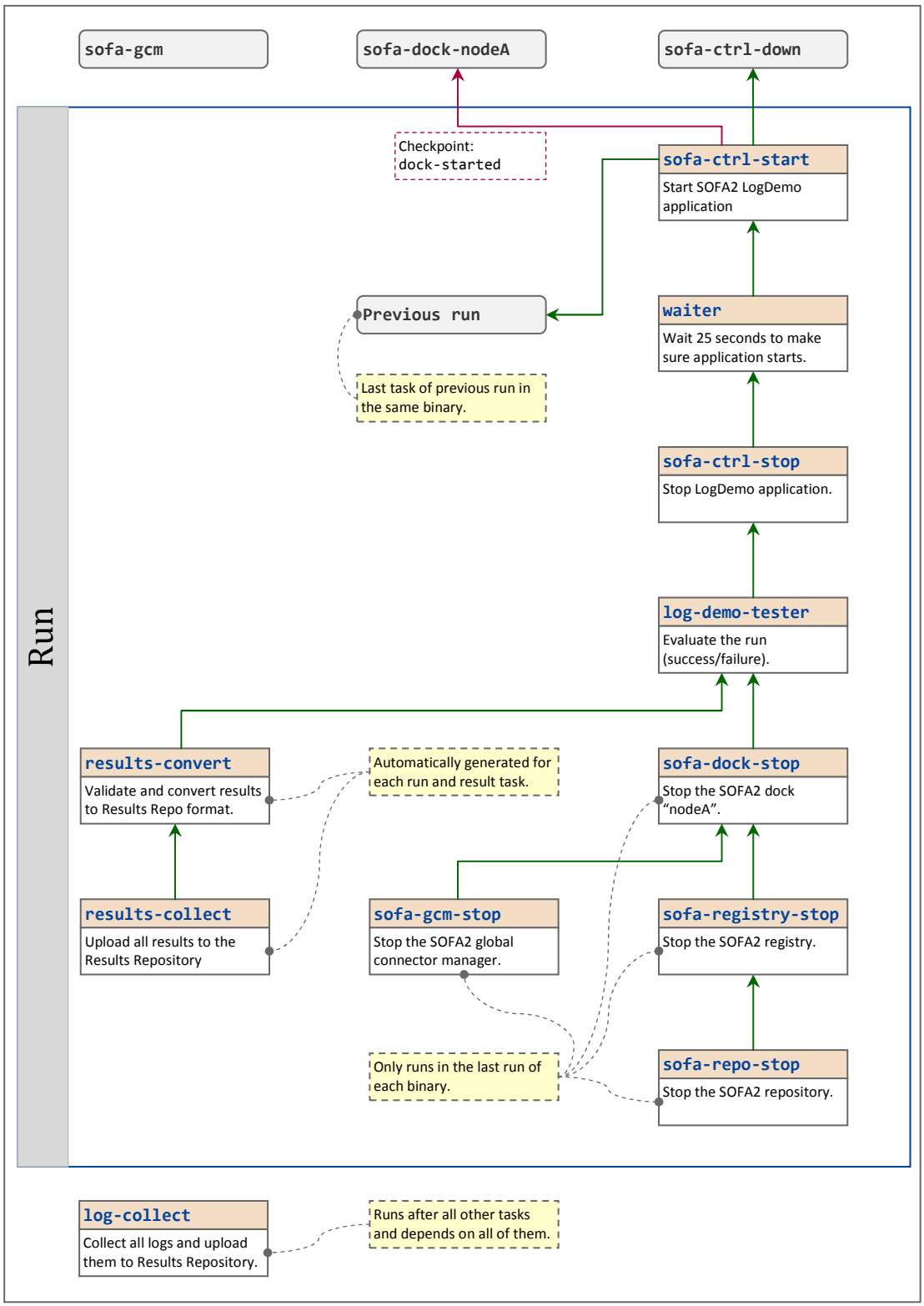


Figure 2: SOFA2 LogDemo test tasks scheme. The picture only shows tasks from the „run“ entity.

7.7 Test Results and Statistics

Four different tests are performed as a part of each run of the LogDemo test case described in this chapter. The file with results therefore contains four different rows:

Test column name	Description
start_up	Succeeds if LogDemo started successfully.
output_correct	Succeeds if application output contains "Hello World" text.
no_exceptions	Succeeds if no exception is reported by LogDemo application.
shut_down	Succeeds if LogDemo is terminated successfully.

The result of all experiments can be viewed in Results module in web interface. Results module presents user with hierarchical view of all the entities which have been uploaded to the Results Repository. For each entity different details and statistics is computed and displayed. The exact statistic details or graphs depend on the Benchmark Manager plug-in which was used to generate the data – the statistics scripts are fully customizable.

In the rest of the chapter we will focus on the statistics generated by the Distributed Testing plug-in and provide detailed description alongside the example screenshots displaying the actual results produced by running SOFA2 proof of concept test developed specifically for the thesis.

In addition to the statistics computed by the plug-in, some of the details are displayed for all of the entities and can be accessed via tabs titled *Metadata*, *Logs* and *Export*.

Notation

In the rest of the chapter we will use following notation to describe the statistics calculated by the plug-in:

- $X_i(a, e, b, r, h)$ – is the result of test i performed on host h in run r in binary b in experiment e in analysis a. The value is either 0 (failed) or 1 (succeeded).
- $R(a, e, b)$ – set of all runs in binary b in experiment e in analysis a.
- $H(a, e)$ – set of all hosts in binary b in experiment e in analysis a.
- $B(a, e)$ – set of all binaries in experiment e in analysis a.
- $E(a)$ – set of all experiments in analysis a.
- $T(a, e, h)$ – set of all tests which are performed in analysis a in experiment e on host h. This set is never empty (otherwise the host would not be included in the results).

When referring to the cells in the table we will only take the cells with the data into account. Header rows and header columns will be ignored. Since each table always has exactly one header line and one header column, the $(i, j)^{th}$ cell in the formulas will refer to cell at $(i + 1)^{th}$ line and $(j + 1)^{th}$ column in the table displayed in the Web Interface.

Main Results Screen

Main screen of the results viewer displays all the analyses which have been scheduled in the environment.

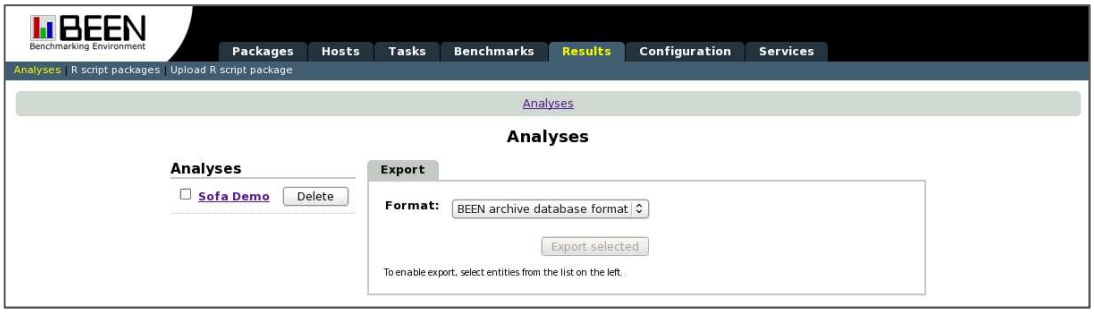


Figure 3: Main results display screen showing all the analyses.

After clicking on selected analysis the Analysis details view is displayed. It is also possible to export all the data BEEN collected for each analysis. If this option is selected, all the analyses selected in the list on the left side are exported and downloaded to the user's computer. Similar functionality is provided for the other entities on the result screens for that entity.

Analysis Details Screen

For each analysis the Results module displays list of all experiments contained within the selected analysis as well as summary statistics for whole analysis.

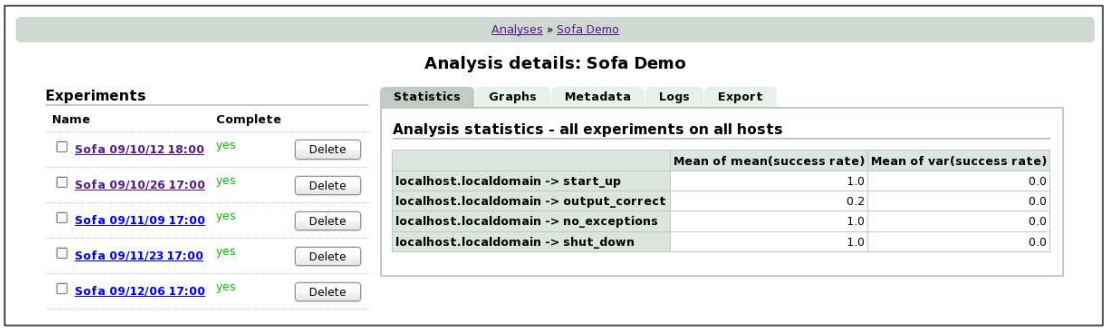


Figure 4: Analysis details screen.

The right-hand side of the screen displays the statistics calculated by the plug-in. Distributed Testing plug-in required at least two successfully finished experiments in an analysis before the statistics can be calculated.
The table generated by the plug-in contains following information:

- **Table rows** – each row shows statistics for all the tests performed on all hosts in the environment. Since multiple result-generating tasks can run on different hosts, the full host name as well as the test name is shown in the first column.
- **Table columns** – the table will always contains only two data columns. The first column titled "*Mean of mean(success rate)*" displays the mean value of all means of experiment success rates for all the experiments. Second column – "*Mean of var(success rate)*" – displays mean value of variances of experiment's success rate for all experiments.

That is, for n-th test (n-th line in the table) the column values can be calculated as:

$$M_n = \underset{\forall e \in E(a)}{\text{mean}} \left(\underset{\substack{\forall b \in B(a,e) \\ \forall r \in R(a,e,b)}}{\text{mean}} X_i(a,e,b,r,h) \right)$$

$$V_n = \underset{\forall e \in E(a)}{\text{mean}} \left(\underset{\substack{\forall b \in B(a,e) \\ \forall r \in R(a,e,b)}}{\text{var}} X_i(a,e,b,r,h) \right)$$

Where h and i are host index and test index such that (i,h) describe valid *(host, test index)* pair and following holds:

$$i + \sum_{k=1}^{h-1} |T(a,e,k)| = n$$

The above basically means that all the tests from all the hosts are put after each other and grouped according to the host. Test ordering into the rows is defined by simple algorithm which can be descibed by following pseudo-code:

```
test_names = []; // empty list
for H in 1 to hostCount
    for each test T on host H
        test_names.append(getHostName(H) + " -> " + getTestName(H, T))
```

Ordering of hosts within the test depends on the experiment's TDL code. Ordering of tests on given host is the same as is the order of tests in the result-generating task which runs on given host.

For analysis a graph displaying success rate of each experiment and each test is plotted as well.

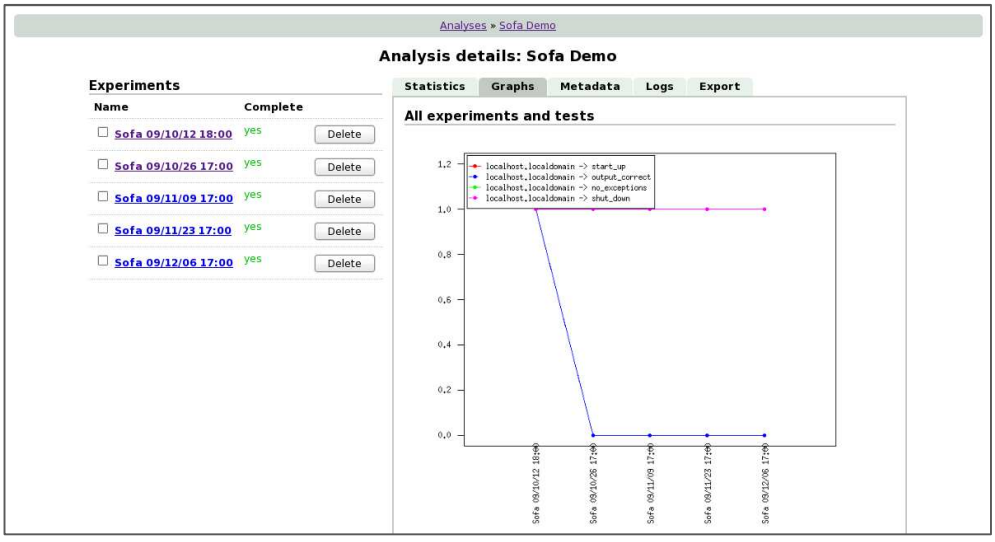


Figure 5: Analysis details screen.

On the horizontal axis the experiments are displayed while vertical axis show the success rate of all runs in given experiment. Different colours are assigned to different tests.

Experiment Details Screen

After clicking on any of the experiments in the analysis details screen, the details of the experiment are shown.

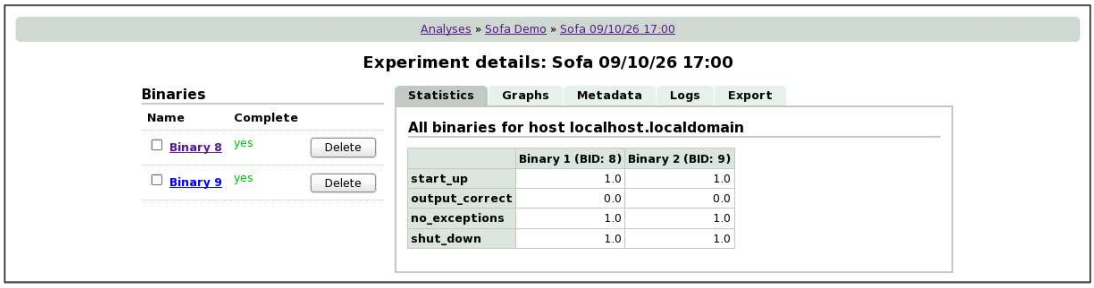


Figure 6: Experiment details screen – statistics view.

As in the case of analysis display, the left-hand side displays the list of entities on next level of the experiment hierarchy – binaries within the current experiment.

For experiment multiple tables can be generated in cases where the tasks which collect results run on multiple hosts. Summary for each host is displayed in separate section in separate table.

The table for each host contains following information:

- **Table rows** – each row shows details for given test performed on given host. In this case it is not necessary to include host name in the row description field as it is mentioned in the section title and only the test name is shown. Tables for different hosts do not need to have the same number of rows – if some result generating task performs more tests, the table for its host will have more rows.
- **Table columns** – each column shows mean success rate of given test on given host across all runs in the binary. Column title contains index of the binary as well as BID of the binary (BID will be one of Ids displayed as binary names on the left side of the screen). All the tables will have the same number of columns if all the binaries have been finished.

The value of $(i,j)^{th}$ element of the table for host h in experiment e in analysis a can be calculated with following formula:

$$M_{i,j} = \underset{\forall r \in R(a,e,j)}{\text{mean}} X_i(a,e,j,r,h)$$

In addition to the statistical details displayed in the tables, the plug-in also plots graph with overview of all the runs in the experiment. The graph can be accessed after clicking on the *Graphs* tab.

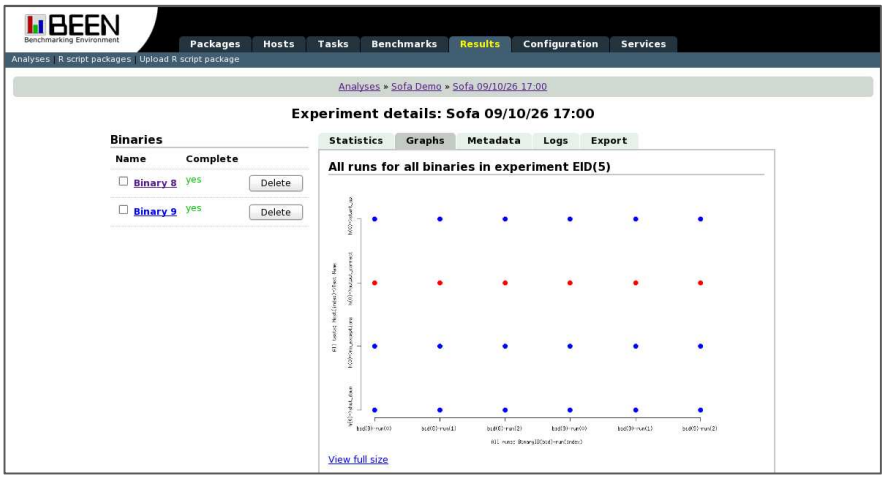


Figure 7: Experiment details screen - graph preview.

The dots in the graphs show the result of each particular test in each run – blue dots are for successful tests, red dots denote failed tests.
The axes in the graph contain following information:

- **X axis** – shows all runs in all binaries in the experiment. The label contains BID of the binary and index of the run within the binary.
- **Y axis** – shows all tests performed on all hosts which run result-generating tasks. Labels contains index of the host with the experiment (full host names would be too long) and name of the test performed.

After clicking on the graph, it is displayed in its full resolution.

Binary Details Screen

After selecting binary from the list of binaries in the experiment the Binary Details screen is shown.

Figure 8: Binary details screen.

The statistics for binary can consist of multiple tables – one table is generated for each host on which result-generating task has been run. Each such table is shown in its own section.

Each binary table contains following information:

- **Table rows** – each row shows results of given test for all runs in the binary. As in the case of binary details screen, the row name is just the name of the test which has been performed on given host. Tables for different hosts do not need to have the same number of rows as this depends on the tests performed on given host.
- **Table columns** – each column shows the results of particular run. Zero values mean failed run, value of one means successful run. Column title contains index of the run within its binary. The index is not run ID and therefore it does not need to correspond to the names of runs displayed in the list of runs on the left side of the screen.

The value of $(i, j)^{th}$ element of h-th table (h is the index of the host) for analysis a, experiment e and binary b can be calculated with following formula:

$$M_{i,j} = X_i(a, e, b, j, h)$$

Run Details Screen

Runs are the lowest level of the entity hierarchy for which the results are displayed. Therefore there is no list of entities shown on the left side.

Figure 9: Run details screen.

For each run multiple tables can be displayed – one table for each host on which result-generating tasks have been run.

Each table will have as many rows as there are tests performed on given host. Tables will always have only one column.

The values in the tables are the results of given test on given host for specified run. Therefore the values can be either 0 for unsuccessful test or 1 for successful test.

7.8 Result Analysis

To make sure the results we have collected and displayed in previous section can be reproduced again, we have modified the default SOFA2 test case to only test specific SOFA versions.

More precisely, the only modification of the test case code was in the Version Provider. The modified test case always creates five experiments two weeks apart from each other. Therefore SOFA2 versions spanning almost two months are tested. Following dates and times are used when checking-out the source code from SVN:

Date and time	Timestamp (ms since Unix epoch)
6. 12. 2009 16:00	1260115200000
23.11.2009 16:00	1258992000000
9.11.2009 17:00	1257782400000
26. 10. 2009 17:00	1256572800000
12. 10. 2009 18:00	1255363200000

Therefore the only difference is that the original version will create new experiments indefinitely while the modified one will always test only pre-selected versions.

Both implementations – the original and the modified one – are attached on the distribution DVD for comparison. The modified version can be found in file called `sofa-fixed-versions.tdl` while the unmodified version is called `sofa-all.tdl`.

All the results have been collected on Fedora 12 32-bit running as a guest inside VMWare virtual machine. Virtual Machine with following configuration has been used:

- Disk drive: 8 GB, SCSI
- Memory: 768 MB
- Network: bridged virtual network adapter (VMWare default settings)

When configuring the experiments we have selected to build two binaries and three runs for each experiment.

From the results it can be seen that between 12th of October and 26th of October a change has been made which broke the LogDemo application since it stopped working between those two versions.

8 Evaluation

8.1 Goals

We have analyzed and implemented distributed regression testing plug-in for Benchmarking Environment.

extended the Benchmarking Environment with support for distributed regression testing. We have verified the validity and usability of the implementation by implementing distributed test for SOFA2 component-based application.

Extending BEEN with Support for Testing

We have designed a new plug-in for Benchmark Manager which adds the support for distributed testing to the BEEN. While this solution is not entirely clean design-wise (since the Benchmark Manager now has two responsibilities – tests and benchmarks), but it allowed us to test the validity of the concept without requiring extensive changes to the whole BEEN.

We have also extended BEEN Execution Environment with support for more advanced features like value passing between tasks via their checkpoints and properties as well as ability to redefine class path for newly scheduled tasks.

Both these features are crucial when working with complex applications and significantly simplify the development of the tasks supporting the tested application.

We have also extended Results Repository to calculate different statistics for regression tests which

Custom Test Definition Language

We have designed and developed a custom interpreted language – Test Definition Language. TDL is the exclusive way of defining new tests in the new Benchmark Manager plug-in.

We have designed the language to be easily modifiable and extensible by using standardized tools to generate the parser directly from language grammar (JavaCC). At the same time, the language core is separated from the rest of the plug-in by an interface which provides access to the services from BEEN to the language runtime. Therefore it is possible to refactor the language core and move it outside of the plug-in thus increasing the scope in which the language can be used.

Proof of Concept Test Implementation

The usability of the developed solution has been demonstrated on a test suite for a LogDemo component application which is part of the SOFA2 distribution package [6].

The test suite is able to download, build, deploy and test the SOFA2 run-time. All the results are collected and analyzed by Results Repository providing quick overview of the test results via Web Interface.

8.2 Future Work

Distributed regression testing is a very complex subject and the testing methodology is always improving. We have demonstrated that our tool can be used to prepare a test case for a distributed component application.

During the development and testing of the tool we have identified several key areas which would warrant further development and research.

User Interface

The user interface for BEEN has been designed as a web application. This has obvious advantages like ease of deployment and usability since the application can be accessed from any system with any reasonably modern browser.

However, web applications impose several restrictions on the way the user interface works. In general, they tend not to support complex interactions between elements on the page (e.g. drag & drop) and interaction between web application and client's computer is usually limited due to security concerns. Even though there are various technologies which try to improve on this aspect of web programming (e.g. AJAX and related technologies) they still cannot provide rich user interface without significant effort.

Therefore, the user interface would benefit from complete rewrite as a regular application. Considering the cross-platform nature of the BEEN project the resulting UI should also support at least Windows and Linux. Several frameworks exist which would allow development of such an interface, for example very popular and mature Equinox OSGi framework [12] by Eclipse Foundation.

Due to the way tasks are managed and implemented in Benchmarking Environment, it is possible to view each task as a black-box entity which takes specific inputs (properties), produces specific outputs (e.g. files, logs) and requires certain tasks to be in pre-defined state (e.g. finished, checkpoint reached) before they can be started. Therefore they can be easily represented in graphical user interface as widgets which can be moved around and connected together to specify task dependencies and data flow.

Test Definition Language

Our testing of the language itself has proven that it is relatively convenient to use, but there are still several areas where the language could be improved.

Currently, the parser is pretty minimal and does not validate all of the code – for example it does not check if given function, variable or constant exists and is available in given scope. This leads to very unpleasant bugs caused by typos in the test case source case which are only found after the test is started in the worst case. The worst case is such bugs are found only in Version Provider stage. Since it is never executed via a direct user interface action, there is nowhere to display the error message but logs. Considering the amount of logs generated by complex experiments can reach thousands of lines, the message may get lost in all the data.

The grammar of the TDL is loosely based on C++ and Java grammar, but does not implement all the features which are quite common in such languages. For example, TDL does not support user-defined functions or function libraries (imports or includes) which would be very useful when working on larger projects.

Appendix A: TDL Grammar

In this section the grammar of the TDL language is described. The grammar is simplified version of the grammar implemented as javacc source in `TDL.jj` file which can be found among the source code of the Distributed Testing plug-in.

The grammar is specified using the common BNF-like notation. Non-terminal symbols are printed in *`blue italic`*, terminal symbols are enclosed in straight quotes `"like this"`. Alternatives are separated by pipe characters ("|"). Symbol repetitions are signified by "*", "+" and "?" characters which denote zero or more occurrences, one or more occurrence and zero or one occurrence.

TDL source code is parsed as a stream of tokens separated by white-space characters (space, tab character or line break). White-space characters are used only as token delimiters and are skipped during the parsing (except when string literals are parsed.)

Parsing of comments is omitted from the grammar for clarity. TDL allows C++-style one line comments which are introduced by two forward slashes (`"//"`) as well as multi-line comments enclosed between `"/*"` and `"*/"`. Nested comments are not allowed in TDL.

`CompilationUnit` is the starting non-terminal of the grammar. For clarity we have split the grammar into two parts – the helper non-terminals part describes common non-terminals used in the main body of the grammar. Helper non-terminals represent basic concepts like numbers, strings or various operators. In the second part of the grammar, the grammar of the source file is described.

Helper non-terminal symbols

```
String ::=
    """ ( RegularCharacter | QuotedSequence )* """

RegularCharacter ::=
    any character symbol except """, "\", new-line, carriage-return

QuotedSequence ::=
    quoted character sequence with escape sequences as in C++

Letter ::=
    lower-case or upper-case letter of the English alphabet

Digit ::=
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Number ::=
    Integer | Float1 | Float2

Integer ::=
    Digit+

Float1 ::=
     ( ( Digit+ "." Digit* ) | ( Digit* "." Digit+ ) ) Exponent?
```

```
Float2 ::=
    Integer Exponent

Exponent ::=
     ( "e" | "E" ) ( "+" | "-" )? Integer

AssignmentOperator ::=
    "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "<<=" | ">>=" | "&=" | "|="
        | "^=" | ".="

EqualityOperator ::=
    "==" | "!="

ConditionalOperator ::=
    "<" | ">" | "<=" | ">="

ShiftOperator ::=
    "<<" | ">>"

AdditiveOperator ::=
    "+" | "-" | "."

MultiplicativeOperator ::=
    "*" | "/" | "%"
```

Main grammar

```
CompilationUnit ::=
     ( StageCodeBlock | VariableOrConstant )*

CodeBlock ::=
    "{" ( CodeBlock | Statement )* "}"

StageCodeBlock ::=
    "#" Identifier
    "{"
     (
        EntityCodeBlock
        |
        ( VariableOrConstant ";" )
    )*
    "}"

EntityCodeBlock ::=
    Identifier CodeBlock

Statement ::=
    SimpleStatement | CompoundStatement

SimpleStatement ::=
    ( VariableOrConstant | Expression | FlowControlStatement) ";"

FlowControlStatement ::=
    "break" | "continue"

CoumpoundStatement ::=
    IfStatement | WhileLoop | DoWhileLoop | ForLoop
```

```
IfStatement ::=
    "if" "(" Expression ")"
    ( CodeBlock | IfStatement )
    ( "else" ( CodeBlock | IfStatement ) )?

WhileLoop ::=
    "while" "(" Expression ")" CodeBlock

DoWhileLoop ::=
    "do" CodeBlock "while" "(" Expression ")" ";"

ForLoop ::=
    "for"
    "("
    ForLoopInitializer
    ForLoopCondition
    ForLoopUpdate
    ")"
    CodeBlock

ForLoopInitializer ::=
    ( "(" Expression | Variable ")" )? ";"

ForLoopCondition ::=
    ( Expression )? ";"

ForLoopUpdate ::=
    ( Expression )?

VariableOrConstant ::=
    Variable | Constant

Variable ::=
    "var" Identifier ( "=" Expression )?

Constant ::=
    "const" Identifier "=" Expression

Expression ::=
    AssignmentExpression | LogicalOrExpression

AssignmentExpression ::=
    Identifier AssignmentOperator Expression

LogicalOrExpression ::=
    LogicalAndExpression ( "||" LogicalAndExpression )*

LogicalAndExpression ::=
    BitwiseOrExpression ( "&&" BitwiseOrExpression )*

BitwiseOrExpression ::=
    BitwiseXorExpression ( "|" BitwiseXorExpression )*

BitwiseXorExpression ::=
    BitwiseAndExpression ( "^" BitwiseAndExpression )*

BitwiseAndExpression ::=
    EqualityTestExpression ( "&" EqualityTestExpression )*
```

```
EqualityTestExpression ::=
    ConditionalExpression ( EqualityOperator ConditionalExpression )*

ConditionalExpression ::=
    ShiftExpression ( ConditionalOperator ShiftExpression )*

ShiftExpression ::=
    AdditiveExpression ( ShiftOperator AdditiveExpression )*

AdditiveExpression ::=
    MultiplicativeExpression
    ( AdditiveOperator MultiplicativeExpression )*

MultiplicativeExpression ::=
    UnaryExpression ( MultiplicativeOperator UnaryExpression )*

UnaryExpression ::=
    ( ( "+" | "-" ) UnaryExpression )
        | PrefixExpression
        | OtherUnaryExpression

PrefixExpression ::=
    ( "++" | "--" ) PrimaryExpression

PostfixExpression ::=
    PrimaryExpression ( "++" | "--" )?

OtherUnaryExpression ::=
    ( ( "!" | "~" ) UnaryExpression ) | PostfixExpression

PrimaryExpression ::=
    ( "(" Expression ")" )
        | FunctionCall
        | ListDefintion
        | LiteralConstant
        | Reference

FunctionCall ::=
    Identifier
    "("
    (
        ( Expression | ModifierFunctionCall )
        (
            ","
            ( Expression | ModifierFunctionCall )
        )*
    )?
    ")"

ModifierFunctionCall ::=
    "@" Identifier
    "("
    (
        Expression
        ( "," Expression )*
    )?
    ")"
```

```
ListDefintion ::=
    "[" ( Expression ( "," Expression )* )? "]"

LiteralConstant ::=
    Number | String | "true" | "false"

Reference ::=
    Identifier

Identifier ::=
    Letter ( Letter | Digit | "_" )*
```

The grammar above defines several operators with following precedence (top rows of the table have higher precedence than lower rows):

Precedence	Operator	Description	Associativity
1	++	Post-increment	Left to right
	--	Post-decrement	
2	!	Logical negation	Right to left
	~	Bitwise complement	
	++	Pre-increment	
	--	Pre-decrement	
	+	Unary plus	
	-	Unary minus	
3	*	Multiplication	Left to right
	/	Division	
	%	Modulus	
4	+	Plus (addition)	Left to right
	-	Minus (subtraction)	
	.	String concatenation	
5	<<	Arithmetic left shift	Left to right
	>>	Arithmetic right shift	
6	<	Less than	Left to right
	>	Greater than	
	<=	Less than or equal to	
	>=	Greater than or equal to	
7	==	Equal to	Left to right
	!=	Not equal to	
8	&	Bitwise and	Left to right
9	^	Bitwise exclusive or	Left to right
10	|	Bitwise (inclusive) or	Left to right
11	&&	Logical and	Left to right
12	||	Logical or	Left to right
13	=	Assignment	Right to left
	+=	Assignment with addition	
	-=	Assignment with subtraction	
	*=	Assignment with multiplication	
	/=	Assignment with division	

Precedence	Operator	Description	Associativity
13 (cont.)	%=	Assignment with modulus	Right to left
	<<=	Assignment with left shift	
	>>=	Assignment with right shift	
	&=	Assignment with bitwise and	
	|=	Assignment with bitwise or	
	^=	Assignment with bitwise exclusive or	
	.=	Assignment with string concatenation	

Appendix B: Installing BEEN

B.1 Requirements

Since most of the BEEN is written in Java, it is theoretically possible to run the environment on any hardware/software platform with support for Java 5.0 or newer.

However, the configuration detector and utilization monitor used by the Host Manager and Host Runtime use native libraries, which are only available for selected platforms. Running BEEN on any other hardware/software platform is still possible, but the features requiring native libraries will not be available [16].

Current release of the BEEN has been tested on following platforms:

- Microsoft Windows Vista with SP2 32-bit
- Microsoft Windows 7 64-bit
- Fedora 12 32-bit

Additionally, following platforms should work as well although they have not been thoroughly tested:

- Microsoft Windows Vista 32 and 64-bit with different service packs
- Microsoft Windows7 32-bit

Original release of BEEN supported additional platforms which may still work, but have not been tested again due to their age (the original version has been released at the end of 2006).

Result Repository is officially supported only on Fedora 12 as the libraries used by BEEN are no longer available in repositories for other Linux distributions and had to be modified to compile on newer platforms.

Required Software

To run basic BEEN services:

- Sun Java JDK 6.0

To run R scripts in Results Repository:

- R 2.1.x or R 2.2.x with NetCDF supporting libraries installed (see installation instructions)

To run and use the Web Interface:

- Apache Tomcat 5.5.x or newer to host the web application
- Mozilla Firefox 2.0 or newer; Internet Explorer 6.0 or newer; Opera 9 or newer to access the Web Interface

To compile BEEN from source:

- Sun Java JDK 6.0
- Ant 1.6.x or newer

To compile native libraries (Load Monitor and Detectors):

- For Windows DLLs Microsoft Visual Studio 2008
- For Linux libraries GCC 3.4.x or newer
- Doxygen 1.4.7 or newer to compile the documentation for native libraries

Other versions of the software mentioned above may work as well, but have not been tested and are not supported.

Some Benchmark Manager plug-ins may require additional sofware. Consult [16] and [1] for the requirements of plug-ins which have not been developed as part of this thesis.

B.2 Installation

Windows

1. Verify that the computer meets the software requirements specified above.
2. Extract the contents of `been.zip` archive to selected directory using Windows Explorer or any other suitable application.

Note that on Windows the Results Repository does not support evaluation of R scripts. To get fully featured environment, it is required to run Results Repository on Linux host. This is the limilation of the original BEEN environment and the libraries which are used to marshall the data between R and Java.

Linux

1. Verify that the computer meets the software requirements specified above.
2. Extract the `been.tar.gz` archive to the target directory via tar command:
3. Install Results Repository pre-requisites.
   3.1. Copy all RPM packages from `install` directory on distribution DVD to any directory on the target system.
   3.2. Log-in as root (or run all following commands via `sudo`) and change to the directory into which the RPM packages have been copied.
   3.3. Install pre-requisite libraries:

```
yum install tcl-8.4.9-3.i386.rpm --nogpgcheck
yum install tk-8.4.9-3.i386.rpm --nogpgcheck
yum install compat-readline43-4.3-3.i386.rpm --nogpgcheck
yum install R-2.1.0-1.bio.fc3.i586.rpm --nogpgcheck

yum install netcdf
yum install netcdf-devel
yum install udunits
yum install udunits-devel

cp /usr/lib/libnetcdf.so /usr/lib/libnetcdf.a
ln -s /usr/include/netcdf/netcdf.h /usr/include/netcdf.h
```

   3.4. Install RNetCDF library in R. This step requires working GCC:

```
R CMD INSTALL RNetCDF_1.2-1.tar.gz
```

3.5. Install SJava library in R. This step requires working G++:

```
R CMD INSTALL -c SJava_0.69_0_fixed4.tar.gz
```

3.6. Update your profile's `~/.bashrc` file to include following environment settings. This step has to be done for all users which will want to run the BEEN (this is not required for User Interface).

```
R_HOME=/usr/lib/R
R_LIBS=${R_HOME}/lib:${R_HOME}/library/SJava/libs
export R_HOME
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${R_LIBS}
export CLASSPATH=/usr/lib/R/library/SJava:${CLASSPATH}
```

Above guide applies to Fedora 12 only. Similar steps need to be taken to install the Results Repository on other Linux distributions.

Web Interface

Web Interface needs to only be installed on the host which runs the Apache Tomcat instance which will host the web application. Following steps are applicable to both Windows and Linux hosts.

1. Verify that the correct version of Apache Tomcat is installed on the target host.
2. Make sure Tomcat is not running when deploying the application.
3. Copy all files from `install/webinterface` directory on distribution DVD to the `webapps/been` directory in the Tomcat installation directory. The target directory may not exist and has to be created in such case.
4. BEEN User Interface defaults to UTF-8 encoding and you may need to modify Tomcat's settings to respect this if you need to use characters other than standard ANSI in the data you enter into the fields in the Web Interface:
   4.1. Open `conf/server.xml` in Tomcat's installation directory in the text editor.
   4.2. Find the `<Connector>` XML element with attribute `port="8080"` and add following attribute to this element:

```
URIEncoding="UTF-8"
```

   4.3. Save the configuration file.

Appendix C: Using BEEN

To use BEEN and run tests or benchmarks, following is required:

- Run Task Manager
- Run Host Runtime on all hosts participating in the environment
- Run and configure Web Interface
- Run all BEEN services via Web Interface

C.1 Task Manager

Task Manager can be started on only one host and has to be the first runnin component of BEEN's runtime.

Windows

1. Open Command Prompt (via link in Start Menu or by typing `cmd` into the Search box in the Start Menu).
2. Navigate to the `bin` directory in BEEN installation directory by using `cd` command.
3. Run `taskmanager.bat` batch file.
4. Output similar to the following should appear in the Command Prompt:

```
Log level: INFO
Could not load configuration file. New configuration file will be
created (with default values set).
Task Manager started...
```

To stop the Task Manager press Ctrl+C in the Command Prompt window where it has been started. Note that the Task Manager should be only terminated after all the Host Runtimes have been stopped.

Linux

1. Open a terminal window.
2. Navigate to the `bin` directory in BEEN installation directory by using `cd` command.
3. Run `taskmanager.sh` shell script.
4. Output similar to the following should appear in the Command Prompt:

```
Log level: INFO
Could not load configuration file. New configuration file will be
created (with default values set).
Task Manager started...
```

To stop the Task Manager press Ctrl+C in the terminal wher it has been started. Note that the Task Manager should be only terminated after all the Host Runtimes have been stopped.

C.2 Host Runtime

One Host Runtime instance has to be started on each host participating in the test or benchmark. Host Runtime can only be started after Task Manager has been successfully started.

Windows

1. Open Command Prompt (via link in Start Menu or by typing `cmd` into the Search box in the Start Menu).
2. Navigate to the `bin` directory in BEEN installation directory by using `cd` command.
3. Run `hostruntime.bat` batch file with parameter specifying name of the host on which the Task Manager is running.
   For example:

```
hostruntime.bat localhost
```

4. Output similar to the following should appear in the Command Prompt:

```
Note: Can't start the RMI registry - another instance is probably
running.
Host Runtime started...
```

To stop the Host Runtime press Ctrl+C in the Command Prompt window where it has been started. All the tasks running on the Host Runtime will be automatically terminated and Host Runtime will unregister itself from the BEEN environment.

Linux

1. Open a terminal window.
2. Navigate to the `bin` directory in BEEN installation directory by using `cd` command.
3. Run `hostruntime.sh` shell script with parameter specifying name of the host on which the Task Manager is running.
   For example:

```
./hostruntime.sh localhost
```

4. Output similar to the following should appear in the Command Prompt:

```
Note: Can't start the RMI registry - another instance is probably
running.
Host Runtime started...
```

To stop the Host Runtime press Ctrl+C in the terminal window where it has been started. All the tasks running on the Host Runtime will be automatically terminated and Host Runtime will unregister itself from the BEEN environment.

C.3 Web Interface and Services

1. Make sure that the Apache Tomcat is running – e.g. by trying to view page `http://hostname:8080/` where `hostname` is the name of the host on which the Web Interface will be running. If you get an error page, start Apache Tomcat in a way appropriate for you system.

2. To access the Web Interface run your preferred browse and go to `http://hostname:8080/been`. The Web Interface should appear.
3. Click on the ***Configuration*** tab in the Web Interface; fill in the ***Task Manager host name*** field and click on ***Save*** button. If the configuration was updated a green bar will appear at the top of the page. If the configuration was not saved successfully an orange bar with error description will appear instead.
4. Click on ***Services*** tab in the Web Interface.
5. Run all the services by filling host name for each of them and pressing ***Start*** button. If the services started successfully green success message will appear. In case of an error, error description message will be shown. In such case you can inspect logs of the failed service by clicking on the ***Logs*** button.

To get additional information about working with the Web Interface consult [16].

References

[1] "Benchmarking Environment Home Page," http://been.ow2.org/, 2006

[2] "BugZilla Project Home Page," http://www.bugzilla.org/

[3] "JUnit Project Home Page," http://www.junit.org/

[4] "Javacc Project Home Page," https://javacc.dev.java.net/

[5] "MantisBT Home Page," http://www.mantisbt.org/

[6] "SOFA2 Home Page," http://sofa.ow2.org/

[7] "The Fractal Project Home Page," http://fractal.ow2.org/

[8] APACHE SOFTWARE FOUNDATION: "Apache Ant Home Page," http://ant.apache.org/

[9] APACHE SOFTWARE FOUNDATION: "Apache Ivy Home Page," http://ant.apache.org/ivy/

[10] BURES, T., HNETYNKA, P., PLASIL, F.: "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model," Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp.40-48, August 2006

[11] ECMA: "ECMAScript Language Specification," 3rd edition, December 1999

[12] ECLIPSE FOUNDATION: "Equinox Home Page," http://www.eclipse.org/equinox/

[13] GOSLING, J, JOY B., STEELE G., BRACHA, G.: "The Java Language Specification," 3rd edition, Addison-Wesley, ISBN 978-0-321-24678-3, June 2005

[14] HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P.: "HP Quality Center Home Page," https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1131_4000_100__

[15] KALIBERA, T., LEHOTSKY, J., MAJDA, D., REPCEK, B., TOMCANYI, M., TOMECEK, A., TUMA, P., URBAN, J.: "Automated Benchmarking and Analysis Tool," in proceedings of VALUETOOLS 2006, Pisa, Italy; © ACM, ISBN 1-59593-504-5, October 2006

[16] LEHOTSKY, J., MAJDA, D., REPCEK, B., TOMCANYI, M., TOMECEK, A., URBAN, J.: "BEEN User Documentation," http://been.ow2.org/documentation/, 2006

[17] MICROSOFT CORPORATION: "DCOM Remote Protocol Specification," http://msdn.microsoft.com/library/cc201989.aspx

[18] PAN, J.: "Software Testing," Carnegie Mellon University, 1999, http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/

[19] R FOUNDATION: "R Project Home Page," http://www.r-project.org/

[20] SUN MICROSYSTEMS, INC.: "Sun Enterprise JavaBeans Technology Home Page," http://java.sun.com/products/ejb/index.jsp

[21] SZYPERSKI, C., GRUNTZ, D., MURER, S.: "Component Software: Beyond Object-Oriented Programming," 2nd edition, Addison-Wesley, ISBN 0-201-74572-0, January 2002

[22] THE C++ STANDARDS COMMITTEE: “ISO/IEC 14882 Standard for Programming Language C++, Working Draft,” ISO/IEC JTC1/SC22/WG21, June 2009

[23] UNIDATA: “NetCDF Format Home Page,” http://www.unidata.ucar.edu/software/netcdf/

[24] WANG, ANDY JU AN; QIAN, KAI: “Component-Oriented Programming,” A Wiley-Interscience, ISBN 978-0-471-64446-0, March 2005